

Fondamenti di Informatica II

A.A. 2007/2008

Algoritmi



Giuseppe Pes & Giuseppe Regina

Indice

• Introduzione alla Documentazione	3
• Documentazione Pratica	4
○ Avvio del Compilatore da riga di comando	4
○ Avvio del Compilatore usando l'interfaccia grafica	5
▪ Uso dei menù	6
▪ Uso dell'editor del codice sorgente	7
▪ Visualizzazione del ByteCode Testuale	8
▪ Finestra Output di Compilazione	8
• Documentazione Tecnica	10
○ Grammatica di J - -	10
○ Considerazioni preliminari	11
○ Il main() e l'uso di stdafx.h	12
○ Analyzer	15
▪ Token.hh / Token.cc	15
▪ FlexLexer.h / Lex.yy.cxx	19
○ Parser	20
▪ Tabella dei Simboli	21
▪ Programma	24
▪ Comandi	26
▪ Espressioni	34
○ Traduttore	
▪ Traduzione dei Comandi	
▪ Traduzione delle Espressioni	

Introduzione alla Documentazione

Il seguente documento, stilato come supporto al Progetto di Informatica II dell'Anno Accademico 2007/2008 da Giuseppe Regina e Giuseppe Pes, si pone l'obiettivo di esplicitare e illustrare quegli aspetti del progetto che, ad un primo e inattento sguardo, potrebbero sembrare oscuri. Successivamente tuttavia verranno toccati, in una circumnavigazione Colombiana, tutte le coste dell'enorme continente rappresentato dal Progetto.

Il Progetto di Informatica II dell'anno accademico 2007/2008 sarà nel seguito della documentazione denominato semplicemente come "Il Progetto".

Il progetto consisteva nella creazione, tramite strutture dati del linguaggio di programmazione C++, di un compilatore per un semplice linguaggio fittizio, direttamente derivato dal Java, battezzato come Java - - (o, più brevemente, Jmm). Ulteriori informazioni sulla grammatica di tale linguaggio sono riportate nell'apposita sezione della Documentazione.

La documentazione inoltre è stata strutturata in maniera tale che il lettore possa saltare immediatamente alla parte interessata, senza dover per forza aver letto tutto nel timore che qualche precedente concetto possa essere stato saltato. Infatti, la prima parte del documento si pone l'obiettivo di servire coloro i quali saranno gli utilizzatori del progetto, la seconda parte invece, di natura più tecnica, ha come scopo l'esplicazione chiara, corretta e coesa dei meccanismi concettuali di programmazione situati dietro il progetto.

Documentazione Pratica

ATTENZIONE: Le immagini contenute all'interno di questa sezione della Documentazione sono state tratte da una versione non definitiva del Progetto. A causa di alcune possibili variazioni compiute nell'ultima fase dello sviluppo alcune immagini potrebbero differire leggermente dalla versione finale del Progetto.

Avvio del Compilatore da riga di comando

Per avviare il compilatore J - - da riga di comando è innanzitutto necessario avere installato, sul proprio PC Windows/Linux, la Java Virtual Machine scaricabile gratuitamente dal sito Sun Microsystems. Inoltre è consigliabile anche possedere il software di traduzione Jasmin, che si occupa del passaggio da Bytecode testuale a Bytecode binario, anche se il possesso di questo software è facoltativo. Infatti l'ultima versione di Jasmin ovvero la 2.3 è stata inclusa nel progetto.

AVVIO DEL COMPILATORE SU SISTEMI WINDOWS 9x/Me/XP/Vista

1. (Solo 9x/Me/XP) Dal menù di avvio di Windows, selezionate ESEGUI
2. Digitate *cmd* nella casella di testo di "Esegui" (in Vista digitatelo nella casella di ricerca del menù di avvio)
3. Si aprirà il Prompt dei Comandi di Windows. Digitate a questo punto *cd* "*<percorso eseguibile>*" e premete INVIO. Percorso eseguibile è la cartella dove avete installato il file.
4. Digitate *JmmCompiler.exe*" – "*<percorso file>*" dove al posto di percorso file andrà digitato il percorso effettivo del file Jmm che si desidera compilare. Al termine digitate INVIO

AVVIO DEL COMPILATORE SU SISTEMI Linux-BASED

1. Aprite il Terminale nella cartella dove si trova il file out del compilatore
2. Digitate *JmmCompiler.out*" – "*<percorso file>*" dove al posto di percorso file andrà digitato il percorso effettivo del file Jmm che si desidera compilare. Al termine digitate INVIO

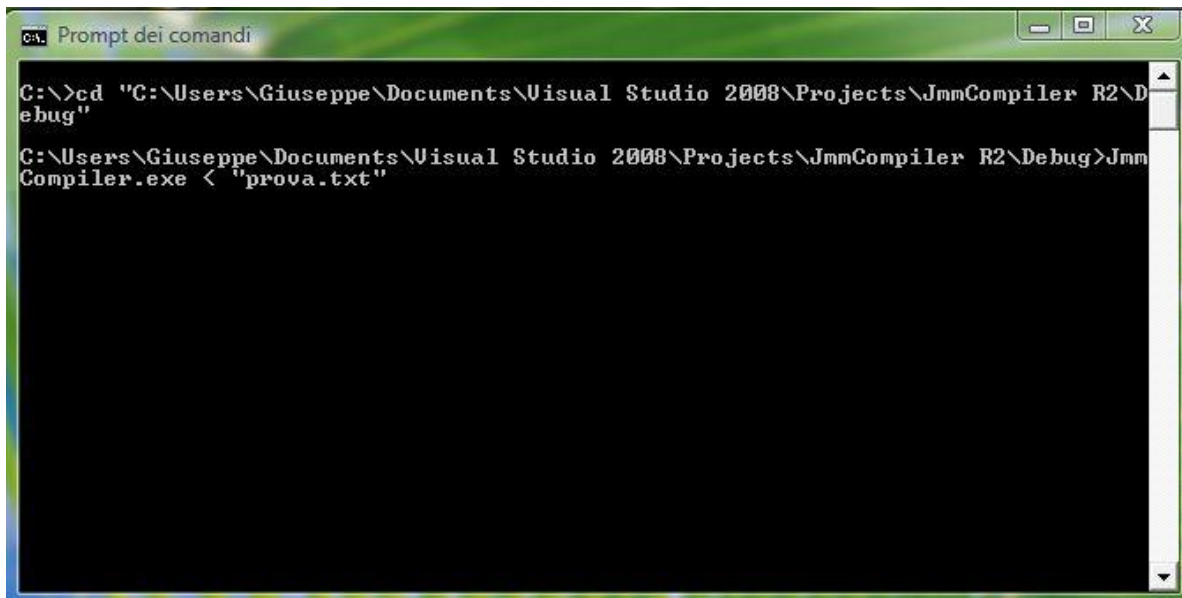


Figura Lancio del Compilatore da riga di comando (Windows)

Avvio del Compilatore tramite l'interfaccia grafica

L'interfaccia grafica per il compilatore J- - è attualmente disponibile solo e soltanto per sistemi Windows con Framework .net 2.0 e seguenti installato su di esso.

AVVIO DEL COMPILATORE SU SISTEMI WINDOWS 9x/Me/XP/Vista

1. Utilizzate Esplora risorse per spostarvi nella cartella contenente l'eseguibile.
2. Lanciate l'interfaccia grafica effettuando doppio clic (o singolo clic a seconda dello stile Windows utilizzato) sull'eseguibile J—GUI.exe

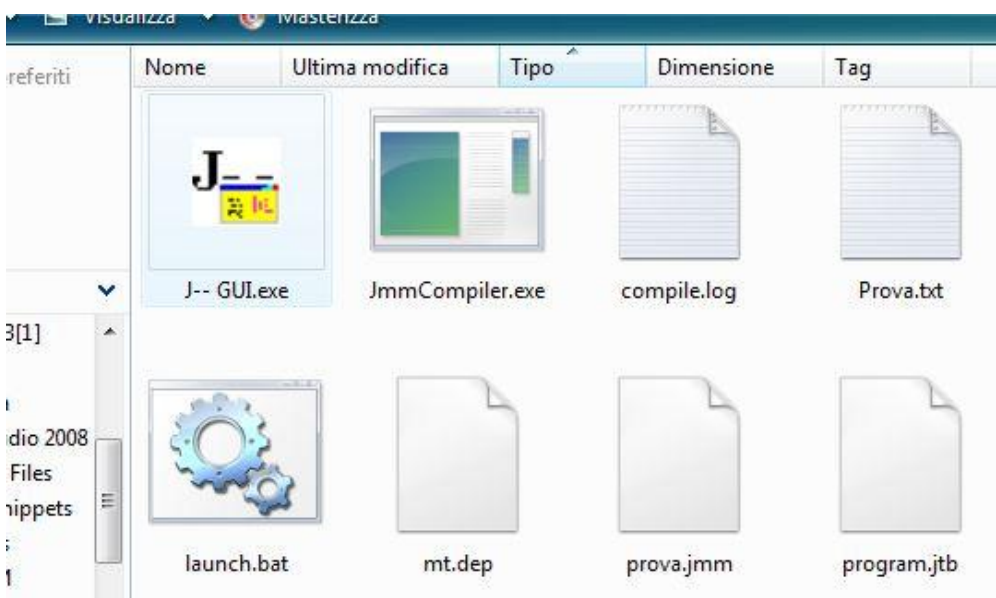


Figura Eseguitibile dell'interfaccia grafica

Uso dei Menù

Fondamentalmente la GUI (Graphic User Interface) di J - - Compiler è composta da due menù, essenzialmente equivalenti ma che, in alcune circostanze, si comportano in modo differente:

- a) Un menù classico a tendina e a discesa
- b) Un menù a striscia (ToolStrip)

Verranno ora esposte le funzionalità di ciascun elemento del menù.













Figura 1 menù di JmmGUI

Menù a Tendina

FILE >	Nuovo Apri Salva Chiudi Esci	Crea un nuovo progetto Apri un progetto salvato Salva il progetto corrente Chiude il progetto corrente Chiude il progetto corrente ed esce dal programma
PROGETTO >	Compila Esegui Compila ed Esegui	Lancia il compilatore e compila il file corrente Esegue l'ultima compilazione del file Compila il file corrente, dopodiché lo esegue
? >	About...	Visualizza una finestra di informazioni

Menù a striscia

	Crea un nuovo progetto (Equivalente a File>Nuovo)
	Apri un progetto salvato (Equivalente a File>Apri)
	Salva il progetto corrente (Equivalente a File>Salva)
	Taglia la porzione di testo selezionata
	Copia la porzione di testo selezionata
	Incolla a partire dalla porzione di testo selezionata, il contenuto degli appunti

	Lancia il compilatore e compila il file corrente (Equivalente a Progetto>Compila)
	Esegue l'ultima compilazione del file (Equivalente a Progetto>Esegui)
	Compila il file corrente, dopodiché lo esegue (Equivalente a Progetto>Compila ed Esegui)
	Mostra questo documento

Uso dell'Editor del Codice Sorgente

Una volta aperta l'interfaccia grafica, la finestra di modifica codice sorgente sarà, per default, disabilitata. Una volta creato un nuovo progetto o aperto un progetto esistente la finestra sarà abilitata alla modifica e il suo contenuto sarà, rispettivamente, vuoto o contenente il testo del file sorgente.

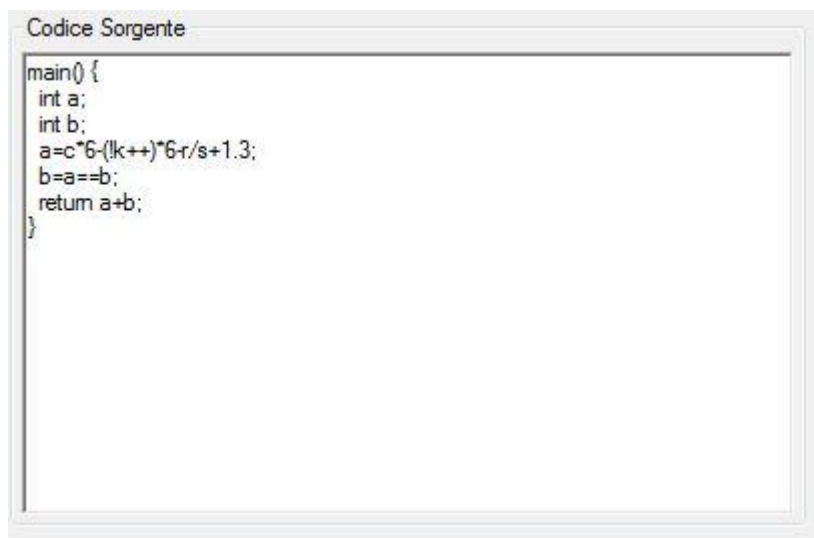


Figura Finestra editor del codice sorgente

La modifica del codice sorgente avviene in maniera analoga a quanto accade per un normale editor di testo: sono quindi consentite le combinazioni classiche di manipolazione di stream come CTRL+X, CTRL+C, CTRL+V oltre che l'uso di un menù a discesa dinamico.

L'unico evento degno di rilevanza che è giusto porre in risalto è la mancanza del riconoscimento delle KeyWord.

Visualizzazione del ByteCode Testuale

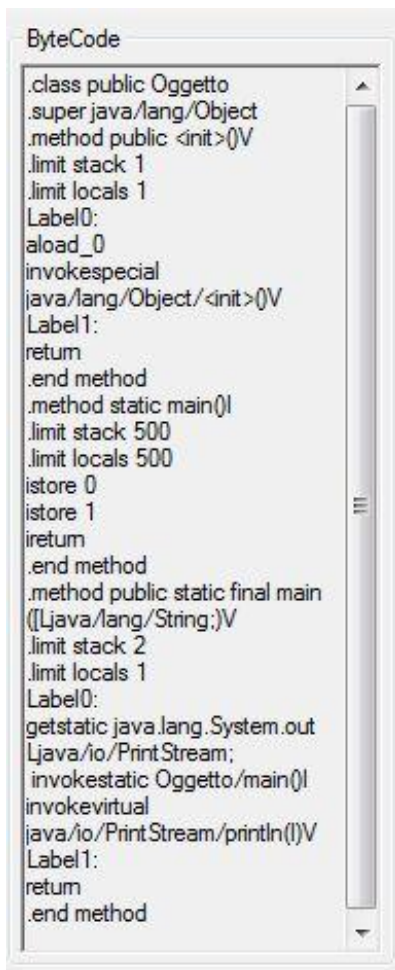


Figura ByteCode Generato

Sebbene JmmCompiler non supporti la modifica in nativo del ByteCode Java, JmmGUI offre la possibilità di visualizzare il codice ByteCode ottenuto con la compilazione. Il codice ottenuto può essere così anche studiato ed analizzato da un punto di vista umano e non meccanico.

Il codice ByteCode dispone della funzione Copia, invocabile tramite la combinazione di tasti CTRL+C ma non dispone né della funzione Taglia, né della funzione Incolla.

Finestra Output di Compilazione

Dopo aver lanciato il compilatore, è possibile che la compilazione non vada a buon fine: in tal caso si viene prontamente avvertiti dalla triplice finestra Output di compilazione presente nella parte inferiore dell'applicazione. La finestra di Output è composta da tre *TabPages* responsabili ciascuna di avvertire l'utente finale in tre diverse situazioni:

- **Informazioni:** avverte l'utente se la compilazione può essere stata rallentata da eventuali troppi programmi in esecuzione o se la JVM non è presente sul computer in uso



Figura Informazioni durante la compilazione

- **Avvisi:** Avverte l'utente se il codice sorgente non contiene alcuna istruzione di Return o se sono state effettuate implicitamente conversioni di tipo.

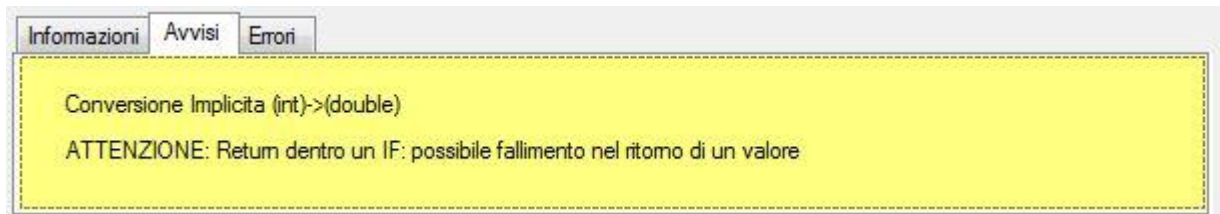


Figura Avvisi durante la compilazione

- **Errori:** Avverte l'utente se la compilazione ha generato errori: variabili non dichiarate, uso invalido di KeyWords e simboli viene segnalato qui.

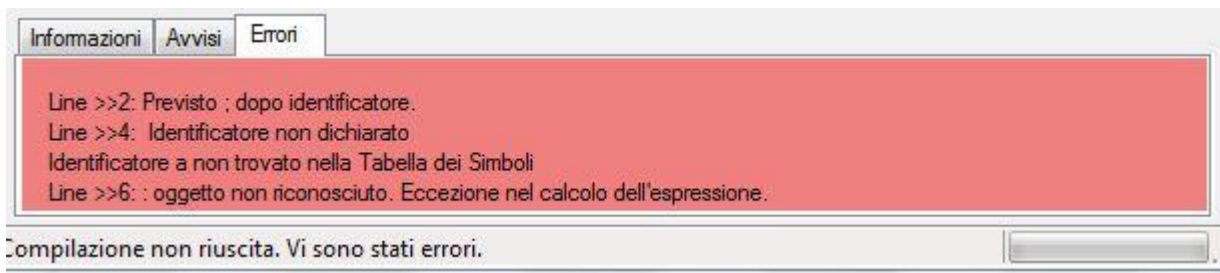


Figura Errori in compilazione

Documentazione Tecnica

ATTENZIONE: Le porzioni di codice contenute in questa sezione della documentazione, eccetto dove diversamente specificato, sono state tutte compilate ed eseguite su Microsoft Visual Studio 2008 Enterprise. Non si assicura la correttezza di compilazione su altri sistemi. Le cause della mancata compilazione possono essere molteplici, da un diverso percorso di inclusione predefinito, ad una mancata corrispondenza tra i nomi dei file. Inoltre, le immagini e i codici sono stati tratti da una versione non definitiva del Progetto (*R:2.0.1 build 6100*). A causa di alcune possibili variazioni compiute nell'ultima fase dello sviluppo alcune immagini e/o porzioni di codice potrebbero differire leggermente dalla versione finale del Progetto.

Grammatica di J - -

Il linguaggio Jmm è un minuscolo sottoinsieme del linguaggio Java che non include gli oggetti ma solo i tipi *int* e *double*. Un programma in linguaggio Jmm è composto da un'unica funzione *main* che restituisce un valore intero. La grammatica del linguaggio Jmm è la seguente:

Program ::= main() {Dec ... Dec Command ... Command}

Dec ::= int I ; | double I ;

Expr ::= Literal | Expr BinOp Expr | I | (Expr) | UnOp Expr

Command ::= I = Expr; | return Expr; | if (Expr) Command else Command | while (Expr) Command | { Command ... Command }

Un programma *Program* è composto da una funzione *main* composta da una sequenza di dichiarazioni (*Dec*) e da una sequenza di comandi (*Command*)

Una dichiarazione è composta da un nome di tipo (*int* o *double*) e da un nome di identificatore.

I valori letterali *Literal* sono costanti intere o reali.

Gli operatori binari *BinOp* e unari *UnOp* supportati dal linguaggio sono i seguenti:

+ * - / operazioni aritmetiche

< > == <= >= confronti

! negazione logica

- inversione

++ pre-incremento

-- pre-decremento

Gli operatori hanno tutti la stessa priorità e gli operatori unari sono tutti prefissi.

I comandi possibili sono:

- assegnamenti a variabili (tramite l'operatore binario =)
- un comando condizionale (*if* con *else*)
- un comando ripetitivo (*while*)
- il *return* che fa terminare il programma restituendo l'espressione
- la sequenza di istruzioni delimitate da { }

La semantica dei comandi e delle espressioni nel linguaggio Jmm è la stessa dei rispettivi comandi e espressioni nel linguaggio C.

Considerazioni Preliminari

Prima di esporre dettagliatamente i singoli componenti del progetto si è deciso di snocciolare rapidamente alcuni parametri di esposizione in maniera tale che il seguito della trattazione possa essere quanto più chiara possibile.

Per tale motivo, si desidera informare il lettore che, data l'enorme mole del codice sorgente, i passi salienti del Progetto verranno esposti per esempi; verrà quindi data una particolare istruzione in linguaggio sorgente Jmm e si vedrà, passo dopo passo, quali sono le istruzioni eseguite e in che modo il Compilatore arriva a produrre codice ByteCode Testuale da affidare all'assemblatore Jasmin.

Inoltre, si è deciso di scrivere gli spezzoni di codice C++ del compilatore col carattere `Courier New` mentre si adotterà il carattere Times New Roman per gli spezzoni di codice Jmm esercitanti la funzione di "esempio-guida".

Infine, si desidera informare il lettore che la guida relativa all'analizzatore lessicale è derivata direttamente dalla guida ufficiale generata da Doxygen. L'unico processo di elaborazione che ha subito è stato quello di essere reimpaginata e riadattata al layout di questo documento.

Il main() e l'uso di stdafx.h

Il main() di JmmCompiler è abbastanza ristretto. Infatti, come più approfonditamente esposto in seguito, si è deciso adottare una struttura Fully-Object-Oriented in maniera da mostrare in superficie solo una minima parte dei processi reali sottostanti.

Di seguito è riportato il codice del main(); tutti i termini in MAIUSCOLO sono macro definite nel file *stdafx.h* (vedi oltre), TokenList è un Typedef a un tipo dati `std::list<Token*>`. Il tipo Token è definito nell'Analyzer (vedi poi).

```
//FILE MAIN.CC

#include "stdafx.h"

int main( int n , char** args ) {
    cout << INTRO << ANALYZE;
    if (n!=2){
        cout << ERRARG;
        system("Pause");
        return 0;
    }
    // prende il file in ingresso
    ifstream fin;
    fin.open(args[1]);
    Analyzer* lexer = new Analyzer(&fin, &cerr);    cout << TLCREATE;
    while(lexer->yylex() != 0); //      Inserisce nella lista
    TokenList& tl = lexer->getTokenList(); //Fa una variabile Tokenlist
    cout << TLOK;
    cout << MAINCH;
    ProgramInitChecker(tl); //controlla il main
    cout << MAINCHOK << ST;
    TokenList::iterator jj = tl.begin();
    TabellaSimboli table(tl,jj); //Fa la tabella dei simboli
    cout << STOK /*<< table*/;
    Program pg(table,tl,jj); //Crea la lista dei comandi
    cout /*<< pg*/ << endl << TL;
    if (!pg.Translate(table)) {
        cerr << CANTRANS;
    }
    else {
        //Jasmine ciene chiamata solo se la traduzione è OK
        cout << TROK << JJ;
        system("java -jar jasmin.jar program.jtb");
        cout << FINALE;
    }
    fin.close();
    delete lexer;
    return 0;
}
```

Come è noto, il linguaggio di programmazione C++, con il quale il Progetto è stato implementato, consente la cosiddetta programmazione a moduli, ossia il programmatore può

distribuire il codice su più file in maniera tale da renderlo più leggibile e funzionale all'aggiornamento.

Tuttavia tale funzione richiede delle *Interfacce* ossia dei file nei quali vi sono le intestazioni delle funzioni implementate.

Per ridurre al minimo il numero di interfacce presenti e per ridurre anche il rischio di doppie inclusioni di file e per non ricorrere ad un numero estremamente elevato di istruzioni del tipo:

```
#ifndef FILE_H
#define FILE_H
/* ... */
#endif
```

Si è deciso di creare un unico file di intestazione comune a tutti i file di intestazione e a tutti i file sorgente. Si è deciso di chiamare questo file *stdafx.h* (STanDard Auto Force eXclusion).

Il corpo del file è il seguente:

```
/*

QUESTO FILE CONTIENE TUTTI I FILE DI INCLUSIONE NECESSARI ALLO SVILUPPO DEL
PROGETTO
POICHE' VERRA' INCLUSO POI UNA SOLA VOLTA NON VI SONO PROBLEMI DI AMBIGUITA' O
DI
RIDEFINIZIONE DI FILE/CLASSI/FUNZIONI.

GENERATO CON MICROSOFT VISUAL STUDIO 2008 ENTERPRISE

*/

#pragma once

//DEFINE

#define INTRO "*****J-- Command Line Compiler (Revision
2)*****\n\n"
#define ANALYZE "Analyzing J-- Sorce Code...\n"
#define TLCREATE "Creating TokenList...\n"
#define TLOK "TokenList succesfully Created!\n"
#define MAINCH "Checking Main(){}..."
#define MAINCHOK "Main(){} Checked!\n"
#define ST "Creating Symbol Table...\n"
#define STOK "Symbol Table succesfully Created!\n"
#define TL "Translating..."
#define CANTRANS "E' stato impossibile tradurre il programma Jmm in ByteCode.
Consultare il file compile.log e translate.log per ulteriori informazioni\n"
#define TROK "Translation Complete!\n"
#define JJ "Transforming JTB in JBB...\n"
#define FINALE "Compilation Accomplished!\n"
```

```

#define ERRARG "Failed to find file.\nE' stato impossibile recuperare il nome
del file dall'elenco degli argomenti\nE' possibile che il numero di argomenti
fosse errato.\nDigitare\n\tJmmCompiler  nomefile.estensione\n"

//    INCLUDE TUTTE LE LIBRERIE C++

#include <string>
#include <cstring>
#include <stdexcept>
#include <fstream>
#include <map>
#include <iostream>
#include <list>
#include <cctype>
#include <sstream>

//    INCLUDE TUTTE LE INTESTAZIONI DELL'ANALYZER

#include "FlexLexer.h"
#include "token.hh"
#include "Analyzer.hh"

//    INCLUDE I SORGENTI DI JmmCompiler R2

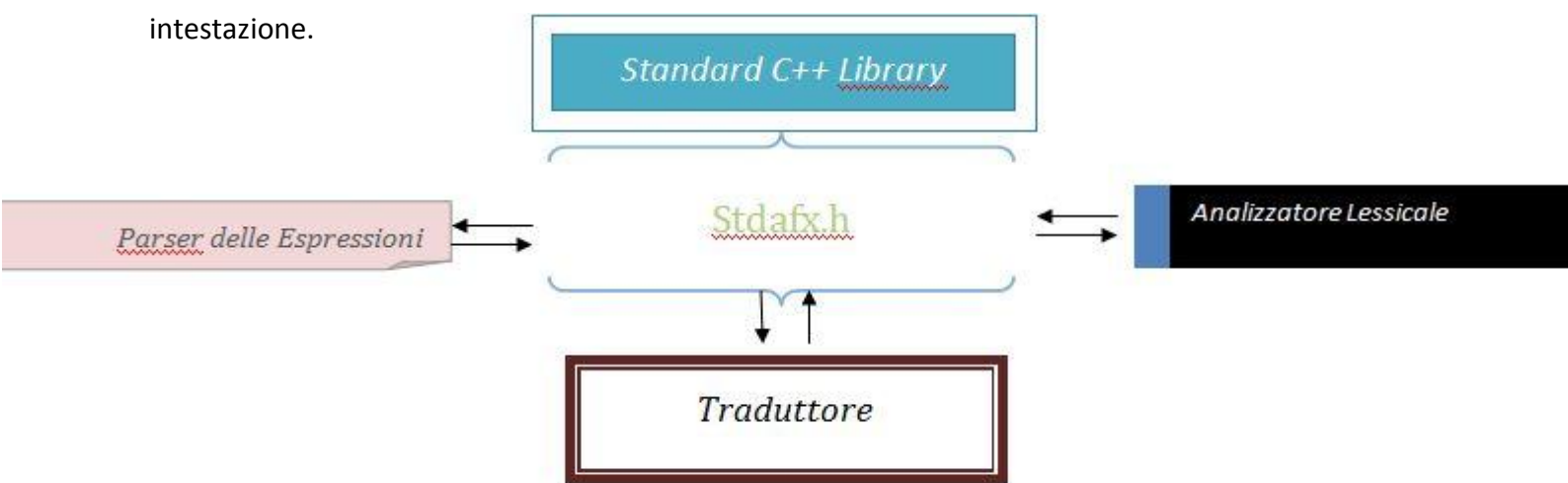
#include "commonFunctions.h"
#include "exception.h"
#include "property.h"
#include "declarations.h"
#include "symboltable.h"
#include "sintassi.h"
#include "exprTree.h"
#include "Program.h"

//    INCLUDE I NAMESPACE

using namespace std;

```

Come si nota facilmente, la struttura del file è a blocchi: si definiscono prima alcune macro (per evitare di definire const globali quelle variabili, in seguito si includono le librerie C++ Standard e in seguito tutti i file del progetto secondo la regola che se un file A richiede la presenza di un file B allora B viene incluso prima di A. L'immagine seguente è esplicativa della struttura del file di intestazione.



Il senso delle frecce infatti rende bene l'idea di una intercomunicazione tra i vari moduli del progetto. Si noti infine che ogni "Entità" rappresenta solo il file sorgente .cpp . Infatti Le intestazioni sono tutte incluse in Stdafx.

Questo potrebbe quindi portare ad una sicura ridefinizione di qualche classe, nonostante l'accorgimento dell'inclusione ordinata, per cui si è inserita la direttiva `#pragma once` che, in ogni caso, include questo file una e una sola volta.

Analyzer

L'analyzer, come si sarà notato dal paragrafo precedente, è il primo oggetto che il main crea, tramite l'istruzione:

```
Analyzer* lexer = new Analyzer(&cin, &cerr);
```

L'oggetto denominato lexer si occupa di prendere il file sorgente e di esaminarlo (tramite il passaggio dello stream di lettura *cin*) e inoltre di generare tramite le istruzioni

```
while(lexer->yylex() != 0);  
TokenList& tl = lexer->getTokenList();
```

L'oggetto TokenList utilizzato successivamente da tutti gli oggetti del Compilatore. Si riporta quindi qui di seguito la documentazione dell'Analyzer, comprese quindi le classi:

- Token
- FlexLexer

Che, come già precedentemente annunciato, è relativa ai file ufficiali creati da *Doxygen*.

Token.hh / Token.cc

Il file Token.hh e, conseguentemente, il file Token.cc contengono classi, metodi e membri dato relativi agli oggetti "base" del linguaggio Jmm. Ogni Token quindi può essere diviso in sottocategorie più specifiche quali Identificatore, Letterale, KeyWord, Operatori e Tipo Dato. La classe Token quindi, per quanto esplicitato poc'anzi, risulta essere una classe virtuale pura dalla quale si derivano poi le classi più specifiche relative ai particolari Token.

Ecco quindi come appare l'intestazione della classe Token:

```
class Token {
protected:
    Token() {};
public:
    int lineNumber;
    virtual ~Token() {};
    virtual bool isIdentifier() { return false; }
    virtual bool isNumber() { return false; }
    virtual bool isSymbol() { return false; };
    virtual bool isKeyword() { return false; };
    virtual bool isDataType() { return false; };
    virtual bool isBinaryOperator() {return false;}
    virtual bool isUnaryOperator() {return false;}
    virtual std::string toString() const = 0;
};
```

Si nota bene come da una parte il costruttore protetto, dall'altra la funzione toString() contribuiscano a rendere rispettivamente la classe virtuale e virtuale pura.

Le funzioni booleane restituiscono tutte ovviamente un valore FALSE. Questo perché un Token generico non è in alcun modo un Token particolare. Le classi derivate ridefiniranno solo e soltanto le funzioni di loro competenza ritornando FALSE negli altri casi.

Ecco quindi l'intestazione (e in alcuni casi la definizioni di alcune funzioni) di tutte le classi che derivano da Token:

```
class Number : public Token {
public:
    DataTypeName t;
    void* number;
    Number(int i) {
        number = new int(i);
        t = INT;
    }
    Number(long l) {
        number = new long(l);
        t = LONG;
    }
    Number(double d) {
        number = new double(d);
        t = DOUBLE;
    }
    Number(bool b) {
        number = new int(b);
        t = INT;
    }
    std::string toString() const;
    ~Number();
    virtual bool isNumber() { return true; }
};
```



```

class Identifier : public Token {
public:
    std::string name;
    Identifier(std::string s) : name(s) {};
    virtual bool isIdentifier() { return true; };
    std::string toString() const ;
};

class DataType : public Token {
public:
    DataTypeName name;
    DataType(DataTypeName n) : name(n) {};
    virtual bool isDataType() { return true; };
    std::string toString() const ;
};

class Keyword : public Token {
public:
    KeywordName name;
    Keyword(KeywordName n) : name(n) {};
    virtual bool isKeyword() { return true; };
    std::string toString() const ;
};

class Symbol : public Token {
public:
    std::string name;
    Symbol(std::string s) : name(s){};
    virtual bool isSymbol() { return true; };
    std::string toString() const;
};

class BinaryOperator : virtual public Symbol {
public:
    BinaryOperator(std::string s) : Symbol(s){};
    virtual bool isBinaryOperator() { return true; };
    std::string toString() const;
};

class UnaryOperator : virtual public Symbol {
public:
    UnaryOperator(std::string s) : Symbol(s){};
    virtual bool isUnaryOperator() { return true; };
    std::string toString() const;
};

```

I metodi, come si evince facilmente da una fugace occhiata al codice, sono abbastanza auto esplicativi e non fanno altro che assegnare i parametri formali dei costruttori ai membri dato della classe o ritornare tali valori. Si noti che le classi BinaryOperator e UnaryOperator non derivano direttamente da Token ma da Symbol. In questo modo, convertendo un oggetto di tale tipo nel tipo madre se ne ottiene la rappresentazione in forma di stringa.

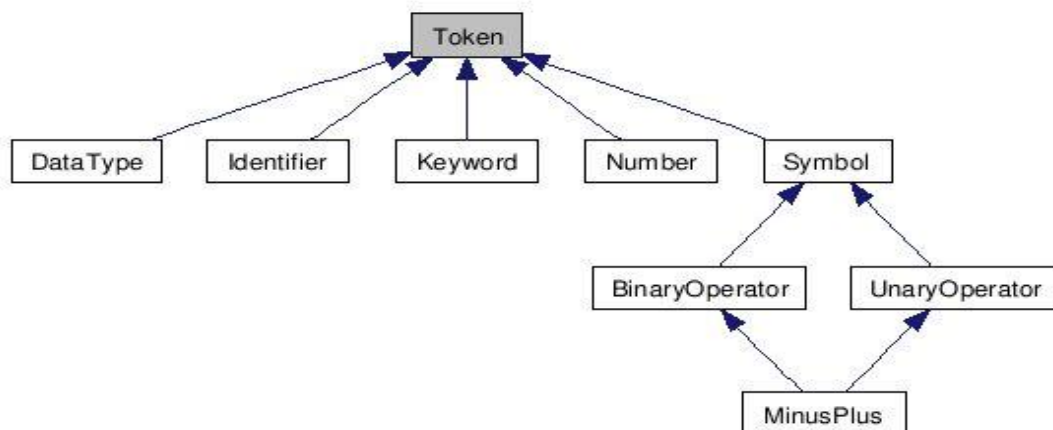
Un discorso a parte invece merita la classe MinusPlus, la cui intestazione è la seguente:

```
class MinusPlus : public BinaryOperator, public UnaryOperator {  
public:  
    MinusPlus(std::string s) : Symbol(s), BinaryOperator(s),  
        UnaryOperator(s) {};  
    std::string toString() const ;  
};
```

si nota infatti come essa erediti sia da BinaryOperator che da UnaryOperator. L'eredità multipla, dal C++ supportata è generalmente un argomento ostico sebbene il principio che giace alla sua base sia semplice: una classe che deriva in maniera multipla da due o più classi eredita sia i metodi della classe madre, sia i metodi della classe padre. Si ha in questo caso un effetto chiamato *Dominanza*. Dalla documentazione MSDN si ha la seguente definizione:

It is possible for more than one name (function, object, or enumerator) to be reached through an inheritance graph. Such cases are considered ambiguous with nonvirtual base classes. They are also ambiguous with virtual base classes, unless one of the names "dominates" the others. A name dominates another name if it is defined in both classes and one class is derived from the other. The dominant name is the name in the derived class; this name is used when an ambiguity would otherwise have arisen.

Tuttavia la presenza della dominanza rende solo e soltanto AVVISI, e non errori, in compilazione. Seppur sconsigliata, la dominanza è lecita e funzionante per cui non vi è null'altro da aggiungere, se non mostrare, come epilogo alla trattazione, lo schema di derivazione delle classi Token.



FlexLexer.h / Lex.yy.cxx

Questi due file, essenziali poiché contenenti la vera e propria definizione e dichiarazione dell'analizzatore lessicale, sono abbastanza complessi nella loro essenza e inoltre, poiché esula dallo scopo di questa documentazione esporne le caratteristiche in maniera approfondita, ci si limiterà ad un semplice sguardo d'insieme:

```
class FlexLexer {
public:
    virtual ~FlexLexer()      { }
    const char* YYText() const { return yytext; }
    int YYLeng()      const { return yyleng; }
    virtual void yy_switch_to_buffer( struct yy_buffer_state* new_buffer )= 0;
    virtual struct yy_buffer_state* yy_create_buffer( FLEX_STD istream* s, int
size ) = 0;
    virtual void yy_delete_buffer( struct yy_buffer_state* b ) = 0;
    virtual void yyrestart( FLEX_STD istream* s ) = 0;
    virtual int yylex() = 0;
    int yylex( FLEX_STD istream* new_in, FLEX_STD ostream* new_out = 0 )
    {
        switch_streams( new_in, new_out );
        return yylex();
    }
    virtual void switch_streams( FLEX_STD istream* new_in = 0,
        FLEX_STD ostream* new_out = 0 ) = 0;
    int lineno() const      { return yylineno; }
    int debug() const       { return yy_flex_debug; }
    void set_debug( int flag ) { yy_flex_debug = flag; }
protected:
    char* yytext;
    int yyleng;
    int yylineno;
    int yy_flex_debug;
};
```

La prima cosa che risalta è la presenza di innumerevoli metodi virtuali nonché di un paio di metodi virtuali puri: la classe è infatti solo una struttura di base dalla quale viene ereditata la classe vero cuore dell'analizzatore.

Il file lex.yy.cxx invece contiene molti metodi di accesso alla memoria scritti in C. E' infatti non rara la presenza di funzioni quali *malloc*, *ralloc* e *free*, come si nota dal codice:

```
new_size = (yy_start_stack_depth) * sizeof( int );
if ( ! (yy_start_stack) )
    (yy_start_stack) = (int *) yyalloca(new_size );
else
    (yy_start_stack) = (int *) yyrealloc((void *)
(yy_start_stack),new_size );
```

Parser

Proseguendo nell'analisi del `main()`, si nota come vi è una chiamata alla funzione `ProgramInitChecker` che si occupa di verificare che un programma abbia una corretta intestazione (`main()`) e conclusione(`}`)

```
void ProgramInitChecker(TokenList& t){
    const string CONTROL_ARRAY[5] = {"main", "(", ")", "{", "}";
    TokenList::iterator it=t.begin();
    TokenList::iterator j=t.end();
    for (int i=0; i<5; i++){
        try {
            if (i!=4) {
                if (it==t.end()){
                    throw new
UnexpectedEndOfProgramException(1,CONTROL_ARRAY[i]);
                }
                if ((*it)-
>toString().find(CONTROL_ARRAY[i])==string::npos) {
                    it++;
                    throw new NotMainException(CONTROL_ARRAY[i]);
                }
                it++;
            }
            else {
                j--;
                if ((*j)-
>toString().find(CONTROL_ARRAY[i])==string::npos) {
                    throw new NotMainException(CONTROL_ARRAY[4]);
                }
                t.erase(--t.end());
            }
        }
        catch (Exception* ex) {
            ex->toString();
        }
    }
    deleteIntro(t,it);
}
```

La funzione definisce un array statico di elementi string e cerca questi elementi nella rappresentazione sottoforma di stringa del Token ottenuta dalla funzione `toString()` della classe Token stessa. Se tale ricerca restituisce un valore const di tipo `string::npos` allora tale stringa non è presente e ciò implica che il main non è inizializzato correttamente.

A titolo di esempio ecco ciò che viene mostrato in alcuni file aventi main non corretto:

<u>Codice J - -</u>	<u>Risultato</u>
[file vuoto]	<pre>*****J-- Command Line Compiler (Revision 2)***** Analyzing J-- Sorce Code... Creating TokenList... TokenList succesfully Created! Checking Main(<>)...Line >> 1 : Raggiunta inaspettatamente la fine del programma Non si e' trovato alcun main La compilazione verrà interrotta.</pre>

main({ })	Analyzing J-- Sorce Code... Creating TokenList... TokenList succesfully Created! Checking Main(<>)...Il programma J-- deve obbligatoriamente iniziare con main(<>{ e finire con: } Carattere non trovato: (< Il programma J-- deve obbligatoriamente iniziare con main(<>{ e finire con: } Carattere non trovato: (< Main(<>{> Checked!
main(){	*****J-- Command Line Compiler (Revision 2)***** Analyzing J-- Sorce Code... Creating TokenList... TokenList succesfully Created! Checking Main(<>)...Il programma J-- deve obbligatoriamente iniziare con main(<>{ e finire con: } Carattere non trovato: } Main(<>{> Checked!

Degno di nota è il fatto che se il file è vuoto la compilazione viene interrotta totalmente. In caso contrario essa continua cercando errori anche nella dichiarazione delle variabili (vedi sezione apposita) o nella sintassi dei comandi (trattata anch'essa in seguito).

Tabella dei Simboli

Successivamente, nel main(), si osserva che l'oggetto che viene creato dopo aver controllato l'intestazione e la conclusione del codice è un oggetto chiamato TabellaSimboli. Tale oggetto rappresenta, in una mappa, il nome di ogni variabile dichiarata correttamente, la linea a cui essa è stata dichiarata e il tipo con cui è stata dichiarata. Tale oggetto è un'istanza di classe Identificatore, derivata da Dichiarazione. Tale artificio permette l'aggiunta di una dichiarazione di funzione, essendo la classe funzione già scritta anche se, al momento di scrittura di questo documento, vuota.

La riga di codice che crea la Tabella dei Simboli è la seguente:

```
TabellaSimboli table(tl,jj); //Fa la tabella dei simboli
```

L'intestazione della classe invece è questa:

```
typedef const string Key_name;

class TabellaSimboli{
private:
    map <Key_name, Dichiarazione *> Mappa_simboli;
    bool isComma( TokenList::const_iterator i);
public:
    TabellaSimboli( TokenList&,TokenList::iterator&);
    TabellaSimboli(const TabellaSimboli &);
    bool present(Key_name&);
    const Dichiarazione* operator[] (Key_name&);
    friend ostream& operator << (ostream&, const TabellaSimboli&);
    ~TabellaSimboli();
};
```

Tralasciando il costruttore di copia che altro non fa che richiamare il costruttore di copia di <map> e tralasciando anche l'operator[] che non fa altro anch'esso di richiamare l'omonimo operatore associato alla mappa e, in ultimo tralasciando l'operator<< che stampa nome, linea e

tipo di ogni variabile, è più opportuno e più remunerativo da un punto di vista strettamente tecnico, considerare il costruttore, la funzione membro `present` e il distruttore.

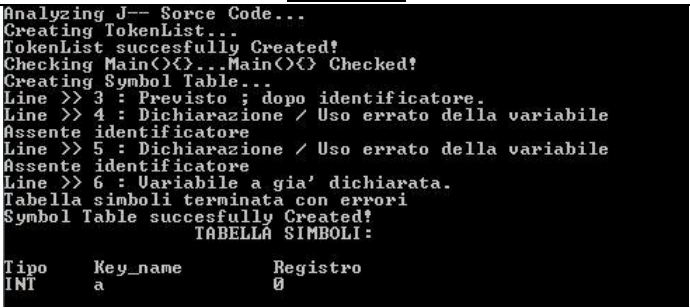
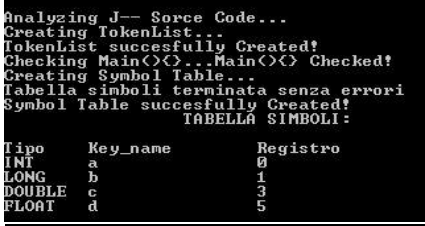
Il costruttore si presenta in questo modo:

```
TabellaSimboli::TabellaSimboli( TokenList& token_list,TokenList::iterator& i){
    bool errore = false;
    while (i != token_list.end() && (*i)->isDataType()) {
        try {
            DataTypeName tipo = ((DataType *) (*i))->name;+
            int linea_dichiarazione = (*i)->lineNumber;
            i++;
            if (i != token_list.end() && (*i)->isIdentifier()) {
                Key_name nome = ((Identifier *) (*i))->name;
                if (present(nome)) {
                    i++;
                    if (i != token_list.end() && isComma(i)) {
                        i++;
                    }
                    errore = true;
                    throw new
DoubleDeclarationException(linea_dichiarazione, nome);
                }
                i++;
                if(i != token_list.end() && isComma(i)) {
                    Mappa_simboli[nome] = new Identificatore;
                    Identificatore * Ident =
((Identificatore*)Mappa_simboli[nome]);
                    Ident->set_nome(nome);
                    Ident->set_tipo(tipo);
                    Ident-
>set_linea_dichiarazione(linea_dichiarazione);
                    Ident->inserisci_in_memmoria();
                    i++;
                }
                else {
                    throw new NoCommaException(linea_dichiarazione);
                    errore = true;
                }
            }
            else {
                if(isComma(i)){
                    i++;
                }
                throw new NoIdentifierException(linea_dichiarazione);
                errore = true;
            }
        }
        catch(Exception* ex) {
            ex->toString();
        }
    }
    token_list.erase ( token_list.begin(), i);
    if (errore) cout << ("Tabella simboli terminata con errori\n");
    else cout << ("Tabella simboli terminata senza errori\n");
}
```

Data la `TokenList`, il costruttore imposta un booleano a false. Successivamente effettua il controllo sul primo Token. Se esso è un tipo dato (`double`, `int`, `long` o `float`) allora la sintassi è corretta e cerca in seguito un identificatore. Se viene trovato si controlla tramite la funzione `present` se esso è già stato dichiarato: in caso affermativo viene lanciata un'eccezione di

tipo `DoubleDeclaration`, altrimenti viene controllata la presenza del `;` (funzione `iscomma()`) e in seguito l'elemento viene inserito regolarmente.

Di seguito si riportano dei codici di esempio con le relative schermate di successo o di errore.

<u>Codice J - -</u>	<u>Risultato</u>
<pre>Main(){ Int a; int b int a; }</pre>	 <pre>Analyzing J-- Source Code... Creating TokenList... TokenList succesfully Created! Checking Main()...Main() Checked! Creating Symbol Table... Line >> 3 : Previsto ; dopo identificatore. Line >> 4 : Dichiarazione / Uso errato della variabile Assente identificatore Line >> 5 : Dichiarazione / Uso errato della variabile Assente identificatore Line >> 6 : Variabile a gia' dichiarata. Tabella simboli terminata con errori Symbol Table succesfully Created! TABELLA SIMBOLI: Tipo Key_name Registro INT a 0</pre>
<pre>main(){ int a; long b; double c; float d; }</pre>	 <pre>Analyzing J-- Source Code... Creating TokenList... TokenList succesfully Created! Checking Main()...Main() Checked! Creating Symbol Table... Tabella simboli terminata senza errori Symbol Table succesfully Created! TABELLA SIMBOLI: Tipo Key_name Registro INT a 0 LONG b 1 DOUBLE c 3 FLOAT d 5</pre>

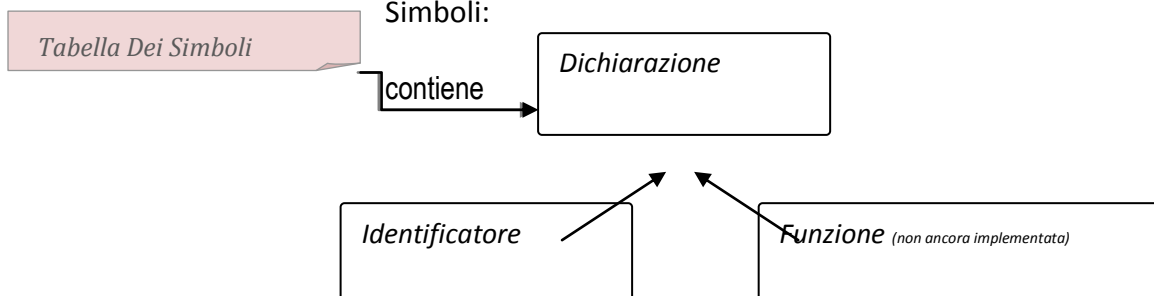
Per completezza ecco il codice della funzione `iscomma()`

```
bool TabellaSimboli::isComma( TokenList::const_iterator i) {
    if ( (*i)->isSymbol() && ((Symbol *) (*i))->name == ";" )
        return true;
    return false;
}
```

E quello del distruttore che non fa altro che eliminare ogni oggetto di tipo `Dichiarazione` dalla mappa.

```
TabellaSimboli::~~TabellaSimboli() {
    map<Key_name,Dichiarazione *>::iterator i =Mappa_simboli.begin();
    for (;i != Mappa_simboli.end(); i++) {
        delete (i->second); // rende l'oggetto che ha immagzinato
    }
    Mappa_simboli.clear();
}
```

Al termine della chiamata al costruttore della Tabella dei Simboli quindi, si ha un oggetto pronto all'uso nella creazione dell'albero sintattico. Prima di passare all'esposizione della classe `Program` tuttavia si presenta qui l'albero di derivazioni della Tabella dei Simboli:



Programma

Proseguendo nell'analisi del `main()` ci si accorge che viene creato, subito dopo la tabella dei simboli, un altro oggetto chiamato `Program`, tramite l'istruzione seguente:

```
Program pg(table,tl,jj); //Crea la lista dei comandi
```

tale oggetto è importantissimo poiché crea l'albero sintattico del programma `Jmm` che si vuole compilare.

L'intestazione della classe `Program` è la seguente:

```
class Program {
private:
    static std::string PRELUDE;
    static std::string ENDING;
    list<Command*> cmd;
    void deleteToken(TokenList&,TokenList::iterator&);
public:
    Program(TabellaSimboli&,TokenList&,TokenList::iterator&);
    bool Translate(TabellaSimboli&);
    friend ostream& operator<<(ostream&,Program&);
    ~Program(void);
};
```

come si nota essa è molto breve e scarna ma tuttavia efficiente e funzionante. I due membri dato statici `string`, `PRELUDE` e `ENDING`, sono costanti e servono alla JVM come preintestazione del file oggetto che sarà generato. La spiegazione di tali stringhe è pertanto non necessaria e quindi inesistente in questo documento.

Il membro dati `cmd` invece rappresenta la sequenza (e quindi la lista) dei comandi del programma. Una spiegazione della classe `Command` è riportata nel seguito.

Anche qui l'operatore di uscita non fa altro che stampare i nomi dei comandi tramite apposite istruzioni della classe `Command` e quindi non verrà esaminato.

Interessante è invece il codice della funzione `Translate` (che sarà trattato nell'apposita sezione) e quello del costruttore che invece è esposto qui di seguito:

```
Program::Program(TabellaSimboli& ts, TokenList& tl, TokenList::iterator& i){
    for (; i != tl.end(); i++) {
        try {
            if ((*i)->isUnaryOperator() || (*i)->isSymbol() &&
((Symbol*)(*i))->name=="(") {
                Command* expCmd = new ExpressionCommand(tl,i);
                this->cmd.push_back(expCmd);
            }
            else if ((*i)->isIdentifier()){
                i++;
                if (i!=tl.end()){
                    //ok, c'è qualcosa dopo.
                    if ((*i)->isSymbol() && ((Symbol*)(*i))-
>name=="=") {
                        i--;
                        Command* Asseg = new
Assignment(tl,ts,i);
                        this->cmd.push_back(Asseg);
                    }
                    else if ((*i)->isUnaryOperator() || (*i)-
>isBinaryOperator()){
```



```

ExpressionCommand(tl,i);

Command* expCParentesi = new
    this->cmd.push_back(expCParentesi);
}
else {
    int linea_errore = (*i)->lineNumber;
    gotoNextInterrupt(tl,i,"");
    throw new SyntaxError(linea_errore);
}
}
else {
    i--;
    throw new UnexpectedEndOfProgramException((*i)-
>lineNumber,"=", ++, "--");
}
}
else if ((*i)->isKeyword()){
    switch (((Keyword*)(*i))->name){
        case IF:{ //if
            Command* ifc = new IFCommand(tl,ts,i);
            this->cmd.push_back(ifc);
            break;
        }
        case WHILE:{ //while
            Command* wcd = new WHILECommand(tl,ts,i);
            this->cmd.push_back(wcd);
            break;
        }
        case RETURN:{ //return
            Command* rtr = new RETURNCommand(tl,ts,i);
            this->cmd.push_back(rtr);
            break;
        }
        default:{
            throw new InvalidKeyWordException((*i)-
>lineNumber, ((Keyword*)(*i))->name);
        }
    }
}
else if ((*i)->isSymbol() && ((Symbol*)(*i))->name=="{"){
    Command* blk= new BlockCommand(tl,ts,i);
    this->cmd.push_back(blk);
}
else{
    throw new SyntaxError((*i)->lineNumber);
}
}
catch (Exception* ex){
    ex->toString();
}
}
}

```

Il costruttore accetta la TokenList, la tabella dei simboli (per verificare l'uso di variabili non dichiarati) e l'iteratore alla prima istruzione del programma che è stata localizzata dal costruttore della tabella dei simboli: l'iteratore infatti, essendo sempre lo stesso passato per riferimento, rimane perennemente al punto finale raggiunto dall'operazione precedente. Questo metodo assicura principalmente e perfettamente un grado di complessità del compilatore non superiore a $O(n)$.

Comandi

Da questi dati inizia l'analisi. Se il primo Token è un operatore unario allora viene chiamato il costruttore della classe `ExpressionCommand` che si occupa di gestire eventi quali:

```
++a;  
!a;  
--++a;
```

E al termine il Comando viene inserito in coda alla lista. Se viene sollevata un'eccezione da questo costruttore viene catturata dal blocco `Try` del costruttore di `Program`.

Se invece il primo Token è un Identificatore viene controllato il Token successivo. Se esso è un operatore binario o unario, viene nuovamente chiamato il costruttore della classe `ExpressionCommand` per gestire gli eventi del tipo:

```
a++;  
a--;  
a+5; //senza senso ma accettato
```

Se invece dopo l'identificatore vi è un operatore binario = allora viene chiamato il costruttore dell'Assegnamento. Al termine il comando viene inserito in fondo alla lista.

Se il primo Token fosse una parola chiave invece, tale parola chiave viene switchata. Se essa rientra nell'ambito delle parole chiavi accettate allora viene chiamato un apposito costruttore per ogni comando, altrimenti viene sollevata un'eccezione di tipo `InvalidKeyword`

Se il primo Token fosse una { allora verrebbe chiamato il costruttore del blocco, mentre se invece non fosse nessuno dei casi precedenti verrebbe sollevata un'eccezione di tipo `SyntaxError`

Al termine viene ripetuto il ciclo e si controlla di nuovo il primo Token successivo al comando creato (anche qui gli iteratori sono passati per riferimento).

Quando viene raggiunta la fine del programma il ciclo termina e il costruttore termina il suo lavoro.

Ecco una serie di piccoli programmi Jmm con relativa opera di creazione dei comandi:

<u>Codice J - -</u>	<u>Risultato</u>
<pre>Main(){ Int a; b=7; a=5 a=3; else return m; }</pre>	<pre>*****J-- Command Line Compiler (Revision 2)***** Analyzing J-- Source Code... Creating TokenList... TokenList succesfully Created! Checking Main(<>)...Main(<>) Checked! Creating Symbol Table... Tabella simboli terminata senza errori Symbol Table succesfully Created! Line >> 3 Identificatore non dichiarato Identificatore b non trovato nella Tabella dei Simboli Line >> 5: oggetto non riconosciuto. Eccezione nel calcolo dell'espressione. Line >> 5: Errore di Sintassi Ogni riga deve essere o un comando chiave o un assegnamento Line >> 5: Errore di Sintassi Ogni riga deve essere o un comando chiave o un assegnamento Line >> 5: Errore di Sintassi Ogni riga deve essere o un comando chiave o un assegnamento Line >> 6Parola chiave else non valida in questo contesto Variabile non dichiarata</pre>

<pre> Main(){ Int a; A=6; If (a==6) Return a; Else a=10; While (a>0){ a=a-1; } Return a; } </pre>	<pre> ***** Albero Sintattico ***** main(<>{ //DICHIARAZIONI Assegnamento alla variabile a l'espressione 6 5EFBCACC IF < a==6 > RETURN a 5EFBCACC ELSE Assegnamento alla variabile a l'espressione 10 5EFBCACC While a>0 BLOCCO < Assegnamento alla variabile a l'espressione a-1 5EFBCACC > FINE BLOCCO 5EFBCACC RETURN a 5EFBCACC } </pre>
--------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Comandi

Come si è visto durante l'analisi della classe Program vi è un'entità fondamentale che viene chiamata durante la costruzione dell'albero ed è la classe Command. L'intestazione di tale classe si presenta così:

```

class Command{
protected:
  Command(){};
public:
  virtual ~Command()=0;
  virtual ostream& stampa(ostream& os)=0;
  virtual bool Translate(ofstream&,TabellaSimboli&)=0;
};

```

Come si nota, essa è virtuale pura. Da questa classe discendono tutti i comandi specifici della grammatica Jmm ossia ASSIGNMENT, IF, WHILE, RETURN e BLOCK. Verranno esaminati uno per volta dettagliatamente. Tuttavia le immagini precedenti valgono come esempio di avvenuta creazione del giusto comando nel giusto punto del programa.

Proseguendo in ordine, verificheremo prima una classe particolare, la ExpressionCommand che si occupa di gestire quei comandi come:

```

++a;
a++;
a--;
a+5; //senza senso ma accettato

```

Questa classe è infatti la più semplice di tutte. La sua intestazione è la seguente:

```

class ExpressionCommand: public Command{
private:
    Albero_espressione* exp;
public:
    ExpressionCommand(TokenList&,TokenList::iterator&);
    ostream& stampa(ostream& os){
        os << "Eseguo l'espressione: " << endl;
        exp->stampa_espressione();
        return os;
    }
    virtual bool Translate(ofstream&,TabellaSimboli&);
    ~ExpressionCommand();
};

```

come si vede, l'unico membro dati della classe è un `Albero_Espressione`. Questo oggetto, che verrà trattato nel seguito, rappresenta, in un albero binario, l'espressione da eseguire. La funzione `stampa()` ritorna uno stream per riferimento e si comporta un po' come l'operatore `<<`. La funzione `Translate()` verrà trattata nell'apposita sezione. Il corpo del costruttore invece, seppur molto semplice è opportuno che sia mostrato, così come quello del distruttore:

```

ExpressionCommand::ExpressionCommand(TokenList & tl, TokenList::iterator & i) {
    this->exp=new Albero_espressione(tl,i);
}

ExpressionCommand::~ExpressionCommand(){
    delete exp;
}

```

Si nota molto bene come in realtà il vero "oggetto" qui non sia l'`ExpressionCommand` ma l'espressione. L'oggetto `ExpressionCommand` è un intermediario che congiunge i comandi con ciò con cui i comandi lavorano, ossia le espressioni.

Passiamo ora alla classe `Assegnamento`. (Assignment). LA dichiarazione è la seguente:

```

class Assignment : public Command{
private:
    Identifier* Var;
    Albero_espressione* exp;
public:
    Assignment(TokenList&,TabellaSimboli&,TokenList::iterator&);
    ostream& stampa(ostream& os){
        os << "Assegnamento alla variabile " << Var->name;
        os << " l'espressione " << endl;
        exp->stampa_espressione();
        return os;
    }
    virtual bool Translate(ofstream&,TabellaSimboli&);
    ~Assignment();
};

```

Acnhe in questo caso, la funzione `Stampa` è inline e il suo funzionamento è simile a quello della classe `ExpressionCommand`. Contiene poi due membri dato che sono la variabile a cui viene assegnato il valore e l'espressione che le verrà assegnata.

Il suo costruttore è leggermente più lungo ma il suo funzionamento è semplice.

```
Assignment::Assignment(TokenList& t, TabellaSimboli& ts, TokenList::iterator& i){
    string nm = ((Identifier*)(*i))->name;
    if (!ts.present(nm)){
        int line=(*i)->lineNumber;
        gotoNextInterrupt(t,i,";");
        throw new UndeclaredIdentifierException(line,nm);
    }
    Var = new Identifier(nm);
    i++;
    if (i==t.end() || !((*i)->isSymbol() && ((Symbol*)(*i))->name=="=")) {
        gotoNextInterrupt(t,i,";");
        throw new SyntaxError((*i)->lineNumber);
    }
    i++;
    this->exp = new Albero_espressione(t,i);
    exp->Error(ts);
    if (i==t.end() || !((*i)->isSymbol() && ((Symbol*)(*i))->name==";")) {
        int ln=(*i)->lineNumber;
        gotoNextInterrupt(t,i,";");
        throw new NoCommaException(ln);
    }
}
```

Dapprima il costruttore salva il nome dell'identificatore (ricordiamo che è la classe Program ad assicurare la presenza di un identificatore alla posizione determinata dall'iteratore) in una stringa temporanea, dopodichè verifica se tale identificatore è presente nella tabella dei simboli. Se non lo è allora viene sollevata un'eccezione di tipo UndeclaredIdentifier, altrimenti viene salvata la variabile e l'iteratore viene avanzato.

Se si è alla fine del programma o se non si trova un = allora si avanza fino al ; seguente (salto degli errori) tramite la funzione gotoNextInterrupt. Il codice della funzione è il seguente:

```
void gotoNextInterrupt(TokenList& tl,TokenList::iterator& ii,const string& ss){
    TokenList::iterator tmp;
    if (ii!=tl.end()){
        tmp=ii;
    }
    else{
        tmp=--ii;
    }
    while (ii!=tl.end() && (*ii)->toString().find("(" + ss + ")") == string::npos){
        ii++;
    }
    if (ii==tl.end()){
        throw new UnexpectedEndOfProgramException((*tmp)->lineNumber,ss);
    }
}
```

La funzione crea dapprima un iteratore temporaneo e lo assegna all'iteratore attuale o a quello precedente a seconda se si sia o no al termine del programma in maniera da ricavare da questo la linea dove l'eccezione si potrebbe verificare. In seguito avanza cercando (in maniera simile a quanto fa la funzione ProgramInitChecker) la stringa che le viene passata (nella maggior parte dei casi tale stringa è un ;). Al termine la funzione ha portato l'iteratore su quel simbolo.

Se invece la variabile era presente nella Tabella dei Simboli, e il simbolo = è stato trovato, viene salvata l'espressione tramite il costruttore di `Albero_espressione` e il suo valore viene assegnato alla variabile di classe `exp`.

Infine viene controllata la semantica dell'espressione (`exp->Error()`) e la presenza del ; finale.

Al termine del processo il comando assegnamento è stato creato e il costruttore di `Program` lo inserisce in fondo alla lista dei comandi.

Un ulteriore comando implementato è il comando Blocco. Questo comando ha un'intestazione di questo tipo:

```
class BlockCommand: public Command{
private:
    list<Command*> ListOfCommands;
public:
    BlockCommand(TokenList&, TabellaSimboli&, TokenList::iterator&);
    ostream& stampa(ostream& os){
        os << "BLOCCO { " << endl;
        for (list<Command*>::iterator i=ListOfCommands.begin();
i!=ListOfCommands.end(); i++){
            os << "\t" << (*i)->stampa(os) << endl;
        }
        os << "} FINE BLOCCO" << endl;
        return os;
    }
    virtual bool Translate(ofstream&, TabellaSimboli&);
    ~BlockCommand();
};
```

E il costruttore è esattamente identico a quello della classe `Program`. L'unica differenza risiede nel fatto che la variabile di classe `cmd` di `Program` viene chiamata qui `ListOfCommand`. La funzione stampa invece viene chiamata virtualmente su ogni elemento di `ListOfCommand`.

In seguito troviamo il comando IF. Esso è rappresentato da una classe derivata di `Command` che possiede la seguente intestazione:

```
class IFCommand: public Command{
private:
    Albero_espressione* exp;
    Command* ActionTrue;
    Command* ActionFalse;
public:
    IFCommand(TokenList&, TabellaSimboli&, TokenList::iterator&);
    ostream& stampa(ostream& os){
        os << "IF (" << endl;
        exp->stampa_espressione();
        os << ")" << endl;
        os << " \t " << (ActionTrue)->stampa(os) << endl;
        os << "ELSE" << endl;
        os << " \t " << (ActionFalse)->stampa(os) << endl;
        return os;
    }
    virtual bool Translate(ofstream&, TabellaSimboli&);
    ~IFCommand();
};
```

Come era facilmente intuibile, il comando IF contiene tre membri dato: l'espressione che deve essere valutata, il comando da eseguire nel caso tale espressione fosse vera e il comando che invece va eseguito nel caso l'espressione fosse falsa.

Il costruttore di questa classe è il seguente:

```
IFCommand::IFCommand(TokenList& tl, TabellaSimboli& ts, TokenList::iterator& i){
    i++;
    this->exp=new Albero_espressione(tl,i,true);
    exp->Error(ts);
    if (i==tl.end()){
        throw new UnexpectedEndOfProgramException ((*--i)-
>lineNumber, "<Command>");
    }
    if ((*i)->isIdentifier()){
        this->ActionTrue = new Assignment(tl,ts,i);
    }
    else if ((*i)->isKeyword()){
        switch (((Keyword*)(*i))->name){
            case IF:{ //if
                this->ActionTrue=new IFCommand(tl,ts,i);
                break;
            }
            case WHILE:{ //while
                this->ActionTrue=new WHILECommand(tl,ts,i);
                break;
            }
            case RETURN:{ //return
                this->ActionTrue=new RETURNCommand(tl,ts,i);
                break;
            }
            default:{
                throw new InvalidKeyWordException ((*i)-
>lineNumber, ((Keyword*)(*i))->name);
            }
        }
    }
    else if ((*i)->isSymbol() && ((Symbol*)(*i))->name=="{"){
        this->ActionTrue=new BlockCommand(tl,ts,i);
    }
    else{
        throw new SyntaxError ((*i)->lineNumber);
    }
    i++;
    if(i==tl.end() || !((*i)->isKeyword() && ((Keyword*)(*i))->name==ELSE)){
        gotoNextInterrupt(tl,i,";");
        throw new NoElseException ((*i)->lineNumber);
    }
    i++;
    if (i==tl.end()){
        throw new UnexpectedEndOfProgramException ((*--i)-
>lineNumber, "<Command>");
    }
    if ((*i)->isIdentifier()){
        this->ActionFalse = new Assignment(tl,ts,i);
    }
    else if ((*i)->isKeyword()){
        switch (((Keyword*)(*i))->name){
            case IF:{ //if
                this->ActionFalse=new IFCommand(tl,ts,i);
                break;
            }
        }
    }
}
```

```

        case WHILE:{ //while
            this->ActionFalse=new WHILECommand(tl,ts,i);
            break;
        }
        case RETURN:{ //return
            this->ActionFalse=new
RETURNCommand(tl,ts,i);

            break;
        }
        default:{
            throw new InvalidKeywordException((*i)-
>lineNumber, ((Keyword*) (*i))->name);
        }
    }
}
else if ((*i)->isSymbol() && ((Symbol*) (*i))->name=="{"){
    this->ActionFalse=new BlockCommand(tl,ts,i);
}
else{
    throw new SyntaxError((*i)->lineNumber);
}
}
}

```

La prima operazione è quella di eseguire il controllo espressione (la presenza del token IF è infatti implicita poiché controllata da Program).

Se tale operazione va a buon fine viene riconosciuto il comando seguente (in maniera analoga al costruttore di Program). Viene salvato tale comando nella variabile `ActionTrue` e viene controllata la presenza dell'ELSE. In caso affermativo viene controllata la sintassi del comando seguente (nello stesso modo ormai già visto) e il comando IF così ottenuto viene inserito in lista. Se si verifica invece qualche eccezione (mancata presenza dell'else ecc...) viene sollevata l'opportuna eccezione.

Rimangono due comandi da esaminare, il comando WHILE e il comando di RETURN. Per semplicità (poiché è esattamente analogo al comando IF se non per la presenza di un solo comando `ActionVerified`) il costruttore di tale classe verrà omissso.

L'intestazione della classe WHILE è la seguente:

```

class WHILECommand: public Command{
private:
    Albero_espressione* exp;
    Command* ActionVerified;
public:
    WHILECommand(TokenList&,TabellaSimboli&,TokenList::iterator&);
    ostream& stampa(ostream& os){
        os << "While " << endl;
        exp->stampa_espressione();
        os << " \t " << (ActionVerified)->stampa(os) << endl;
        return os;
    }
    virtual bool Translate(ofstream&, TabellaSimboli&);
    ~WHILECommand();
};

```

mentre quella della classe RETURN è:


```

class RETURNCommand: public Command{
private:
    Albero_espressione* exp;
public:
    RETURNCommand(TokenList&, TabellaSimboli& ts, TokenList::iterator&);
    ostream& stampa(ostream& os){
        os << "RETURN " << endl;
        exp->stampa_espressione();
        return os;
    }
    virtual bool Translate(ofstream&, TabellaSimboli&);
    ~RETURNCommand();
};

```

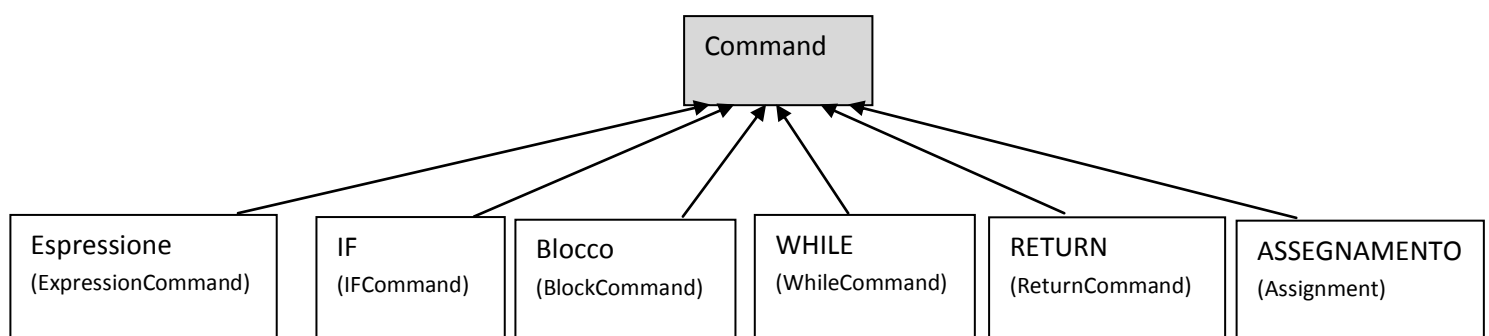
Il costruttore di RETURN è anch'esso semplice:

```

RETURNCommand::RETURNCommand(TokenList& t, TabellaSimboli& ts,
TokenList::iterator& i){
    i++;
    this->exp=new Albero_espressione(t,i);
    exp->Error(ts);
    if (i==t.end() || !((*i)->isSymbol() && ((Symbol*)(*i))->name==";" )){
        throw new NoCommaException((*i)->lineNumber);
    }
}

```

Esso non fa altro che salvare l'espressione da ritornare e immetterla nella variabile di classe exp. I comandi quindi possono essere schematizzati nel seguente diagramma di derivazione:



Espressioni

Si è sicuramente notato, durante l'analisi dei Comandi, che l'oggetto comune a tutte le classi è l'oggetto `Albero_espressione`. Tale oggetto, importantissimo ovviamente, è uno dei più complessi, se non forse il più complesso in assoluto, presente nel compilatore Jmm. Tale descrizione tuttavia pare essere in dissonanza con l'intestazione della classe che, seppur non sia delle più corte, è chiara e limpida anche per chi magari non ha un'eccessiva preparazione in materia:

```
class Albero_espressione{
private:
    struct nodo {
        nodo( nodo * d, nodo * s, Tree_expression * e) : destra(d),
sinistra(s), info(e) {}
        nodo * sinistra;
        nodo * destra;
        Tree_expression* info;
    };
    typedef nodo * nodo_ptr;
    nodo_ptr tree;
    void appendi_operatore(nodo_ptr&, nodo_ptr&, TokenList::iterator&,
TokenList::const_iterator);
    void stampa(nodo_ptr) const;
    void espr_in_albero(nodo_ptr&, TokenList::iterator&,
TokenList::const_iterator);
    nodo_ptr crea_leterale(Number*);
    void stampa_espressione(const nodo_ptr);
    void stampa_posticipata(nodo_ptr);
    void traduci (nodo_ptr, TabellaSimboli&, ofstream&);
    void del (nodo_ptr&);
    bool Error( const nodo_ptr&, TabellaSimboli &TS);
public:
    void stampa() const ;
    Albero_espressione(TokenList&, TokenList::iterator&, bool = false);
    ~Albero_espressione();
    void stampa_espressione(void);
    void stampa_posticipata(void);
    bool traduci(TabellaSimboli&, ofstream&);
    bool Error(TabellaSimboli&);
};
```

I membri dato della classe sono solo due: una struttura che rappresenta il singolo nodo dell'albero e un puntatore a tale struttura (denominato per semplicità tramite un `typedef` come `nodo_ptr`) che rappresenta la radice dell'albero.

La struttura `nodo` contiene un importante membro dati: un puntatore a Classe Base `Tree_expression`. Tale classe base, virtuale pura, rappresenta la generalizzazione di tutto ciò che un albero può contenere: identificatori, letterali, operatori binari ed operatori unari. La si può quindi considerare come una classe parallela alla classe `Token`.

Detto ciò potrebbe sembrare lecito chiedersi come mai si è definita tale classe perché se davvero è una classe parallela a Token allora sarebbe bastato mettere come membro dati anziché un puntatore a Tree_expression, un puntatore a Token. Tuttavia l'uso di tale classe consente una sorta di espansione delle classi derivate da Token senza dover per forza modificare queste ultime. Modifiche effettuate sono ad esempio il tipo di un letterale o la priorità di un operatore. Le dichiarazioni di tutte queste classi sono le seguenti (incluse nel file `sintassi.h`):

```
class Tree_expression {
public:
    virtual bool isLiteral() const { return false;};
    virtual bool isOperatorBinary() const { return false;};
    virtual bool isTree_Identifier() const { return false; }
    virtual bool IsOperatorUnary() const { return false; }
    virtual void stampa() const = 0;
    virtual void traduci( TabellaSimboli&, ofstream&) = 0;
};

class Tree_identifier : public Tree_expression {
    string name;
    DataTypeName tipo;
public:
    Tree_identifier(const string s) : name(s){}
    bool set_tipo(DataTypeName t) { tipo = t; return true; };
    string get_name () { return name; }
    virtual bool isTree_Identifier() const { return true; }
    void stampa() const { cout << name; }
    virtual void traduci(TabellaSimboli&, ofstream&);
};

class Tree_literal : public Tree_expression {
private:
    void* number;
    DataTypeName tipo;
public:
    Tree_literal(int i, DataTypeName t) : tipo(t) { number = new int(i);}
    Tree_literal(float f, DataTypeName t) : tipo(t) { number = new float(f);}
    Tree_literal(double d, DataTypeName t) : tipo(t){number = new double (d);}
    Tree_literal(long int l, DataTypeName t) : tipo(t){number=new long int(l);}
    bool isLiteral() const { return true;};
    void * get_value() const { return number ;}
    DataTypeName get_tipo() const { return tipo;};
    virtual void stampa () const;
    void stampa_file(ofstream&) const;
    virtual void traduci( TabellaSimboli&, ofstream&);
};

class Operatore_unario : public Tree_expression {
    string name;
public:
    Operatore_unario(string op_u) : name(op_u){};
    string get_operator_unary() const { return name ;}
    bool IsOperatorUnary() const { return true; }
    void stampa () const { cout << name ; }
    virtual void traduci( TabellaSimboli&, ofstream&);
};
```

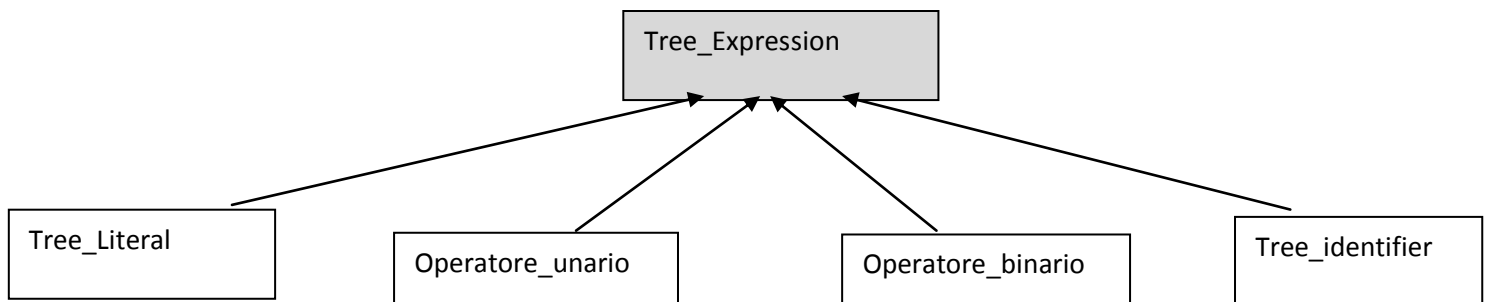
```

class Operatore_binario : public Tree_expression {
private:
    string name;
public:
    Operatore_binario(const string op_b) : name(op_b) {}
    virtual bool isOperatorBinary() const { return true; }
    void stampa() const { cout << name; }
    virtual void traduci( TabellaSimboli&, ofstream& ) ;
};

```

Si nota facilmente la somiglianza con le corrispettive “sorelle” Token: l’uso di funzioni del tipo `isCLASSNAME()` ad esempio o l’uso di funzioni virutali.

Il diagramma di derivazione tuttavia è alquanto diverso in quanto contiene un solo “livello”:



Quando dunque viene chiamato il costruttore delle espressioni (generalmente nei costruttori dei vari comandi) viene eseguito questo codice:

```

Albero_espressione::Albero_espressione(TokenList& tl , TokenList::iterator& i,
bool v) {
    tree = NULL;
    numero_parentesi = 0;
    TokenList::const_iterator end = tl.end();
    Cerca_virgola = v;
    espr_in_albero(tree, i, end);
}

```

A sua volta, come si vede, questo costruttore richiama una funzione ricorsiva: `espr_in_albero()`. La funzione `espr_in_albero()` effettua principalmente due tipi di operazioni, a seconda se il token è o no lecito:

- Se il token è lecito (ossia non vi è errore di sintassi) viene esaminato:
 - Se è un identificatore o un letterale si chiama la funzione `appendi_operatore` per verificare la priorità
 - Se è una parentesi viene richiamata `espr_in_albero()` per salvare tutta l’espressione contenuta tra le parentesi
 - Se è un operatore unario viene effettuato un ciclo (per individuare operatori unari successivi) e al termine viene salvato l’identificatore
- Se il token non è lecito viene lanciato un errore di sintassi

L'attenzione quindi deve spostarsi sulla funzione `appendi_operatore`. Essa ha tre modi di comportarsi a seconda della priorità dell'operatore e dell'essere o meno lecito il token.

- Se è un operatore a priorità alta (* o /) viene legato l'operatore all'identificatore (passato da `espr_in_albero()`) e viene ritornato un nodo vuoto a cui l'operatore è legato. In tale modo l'espressione con l'operatore ad alta priorità si sopsta in basso nell'albero e verrà eseguita prima.
- Se è un operatore a bassa priorità, l'operatore viene legato all'identificatore e l'operatore viene ritornato
- Se il token infine non è lecito viene sollevata una `InvalidExpressionException()`.

Sebbene tali elenchi sembrino estremamente semplici, la realizzazione di queste due funzioni è quella che ha richiesto di più in assoluto durante lo sviluppo del progetto e il suo codice, per quanto gli elenchi possano far credere il contrario, non è affatto banale:

```
void Albero_espressione::espr_in_albero(nodo_ptr& root, TokenList::iterator& i,
TokenList::const_iterator end) {
    if (i == end) {
        if(numero_parentesi > 0) {
            throw new GapException((*i)->lineNumber, ' ');
        }
        throw new NoCommaException((*i)->lineNumber
    }
    else if ((*i)->isSymbol() && ((Symbol *) (*i))->name == ";" ) {
        return;
    }
    else if ((*i)->isSymbol() && ((Symbol *) (*i))->name == ")") {
        numero_parentesi--;
        i--;
        if ((*i)->isBinaryOperator()) {
            throw new NoIdentifierException((*i)->lineNumber);
        }
        i++;
        if (numero_parentesi < 0) {
            throw new InvalidExpressionException((*i)-
>lineNumber, "Parentesi chiusa in eccesso");
        }
        return;
    }
    else if ((*i)->isIdentifier()) {
        string name = ((Identifier *) (*i))->name;
        nodo_ptr nuovo = new nodo( NULL,NULL ,new Tree_identifier(name));
        i++;
        appendi_operatore(root, nuovo, i,end); //
        if (root != NULL) {
            espr_in_albero(root->destra, i, end);
        }
        else {
            root = nuovo;
            return;
        }
    }
    else if ((*i)->isSymbol() && ((Symbol *) (*i))->name == "(") {
        numero_parentesi++;
        nodo_ptr espr = NULL;
        i++;
        espr_in_albero(espr, i, end);
    }
}
```

```

        i++;
        if (Cerca_virgola == true && numero_parentesi == 0) {
            root = espr;
            return;
        }
        appendi_operatore(root, espr, i, end);
        if (root) {
            espr_in_albero(root->destra, i, end);
        }
    }
    else if ((*i)->isNumber()) {
        nodo_ptr new_number = crea_leterale( (Number *) (*i));
        i++;
        appendi_operatore(root, new_number, i, end);
        if (root) {
            espr_in_albero(root->destra, i, end);
        }
        else {
            root = new_number;
        }
    }
    else if ((*i)->isUnaryOperator() ) {
        bool no_more_not=false;
        string type = ((Symbol *) (*i))->name;
        if (type!="!") {
            no_more_not=true;
        }
        nodo_ptr nuovo = new nodo(NULL, NULL, new Operatore_unario(type));
        nodo_ptr op = nuovo;
        i++;
        if (i==end){
            throw new UnexpectedEndOfProgramException ((*--i)-
>lineNumber, "<Identifier>,<UnOp>");
        }
        while ( nuovo != NULL && (*i)->isUnaryOperator() ) {
            if (((Symbol*) (*i))->name != "!") {
                no_more_not=true;
            }
            if (no_more_not && ((Symbol*) (*i))->name=="!") {
                throw new InvalidExpressionException ((*i)-
>lineNumber, "Impossibile usare un ! dopo un <UnOp>");
            }
            string type = ((Symbol *) (*i))->name;
            nuovo->destra = new nodo(NULL, NULL, new
Operatore_unario(type));
            i++;
            nuovo = nuovo->destra;
        }
        if ( (*i)->isIdentifier()) {
            nuovo->destra = new nodo(NULL, NULL, new
Tree_identifier(((Identifier *) (*i))->name));
        }
        else if ((*i)->isNumber()) {
            nodo_ptr new_number = crea_leterale( (Number *) (*i));
            nuovo->destra = new_number;
        }
        else if ((*i)->isSymbol() && ((Symbol *) (*i))->name == "(") {
            numero_parentesi++;
            nodo_ptr espr = NULL;
            i++;
            espr_in_albero(espr, i, end);
            nuovo->destra = espr;
        }
    }
}

```

```

        i++;
        appendi_operatore(root, op, i, end);
        if(root){
            espr_in_albero(root->destra, i, end);
        }
        else {
            root = op;
        }
        return;
    }
    else {
        throw new TypeMismatchException((*i)->lineNumber);
    }
}

void Albero_espressione::appendi_operatore(nodo_ptr& root, nodo_ptr& temp,
TokenList::iterator& i , TokenList::const_iterator end){
    if (i == end) {
        if(numero_parentesi > 0) {
            throw new GapException((*i)->lineNumber, ' ');
        }
        throw new NoCommaException((*i)->lineNumber);
    }
    if ((*i)->isSymbol() && (((Symbol *) (*i))->name == ";" )) {
        if(!root) {
            root = temp;
        }
        return;
    }
    else if ((*i)->isSymbol() && ((Symbol *) (*i))->name == ")") {
        if(!root) {
            root = temp;
        }
        return;
    }
    if ( (*i)->isBinaryOperator()) {
        string type = ((Symbol *) (*i))->name;
        if ( type == "*" || type == "/" ) {
            nodo_ptr nuovo = new nodo(NULL, temp, new
Operatore_binario(type));
            i++;
            if (i==end){
                throw new UnexpectedEndOfProgramException((*--i)-
>lineNumber, "<Identifier>,<UnOp>");
            }
            if( (*i)->isIdentifier()) {
                nuovo->destra = new nodo(NULL, NULL, new
Tree_identifier(((Identifier *) (*i))->name));
            }
            else if ((*i)->isNumber()) {
                nodo_ptr new_number = crea_leterale( (Number *) (*i));
                nuovo->destra = new_number;
            }
            else if ((*i)->isSymbol() && ((Symbol *) (*i))->name == "(") {
                numero_parentesi++;
                nodo_ptr espr = NULL;
                i++;
                espr_in_albero(espr, i, end);
                nuovo->destra = espr;
            }
            i++;
            appendi_operatore(root, nuovo, i, end);
            return;

```

```

    }
    else if ( ((*i)->isBinaryOperator()) && ((Symbol *) (*i))->name ==
"=") {
        throw new InvalidExpressionException ((*i)-
>lineNumber, "Carattere non valido: =");
    }
    else {
        root = new nodo(NULL, temp, new Operatore_binario(type));
        i++;
        return;
    }
}
else if ((*i)->isUnaryOperator() ) {
    string type = ((Symbol *) (*i))->name;
    nodo_ptr nuovo = new nodo(NULL, temp, new Operatore_binario(type));
    i++;
    appendi_operatore(root, nuovo, i, end);
    return;
}
else {
    throw new UnknownObjectException ((*i)->lineNumber, (*i)->toString());
}
}
return;
}

```

Come si nota ad un'attenta analisi vi sono alcune chiamate ad altre eccezioni che possono essere effettuate, ad esempio quando viene effettuato il preincremento/predecremento su un letterale.

Al termine delle chiamate ricorsive, lo stack delle chiamate viene chiuso e si ottiene l'albero nella sua interezza. Tale albero sarà poi assegnato ad una variabile della classe command specifica come ad esempio l'espressione che è nell'assegnamento *et similia*.

Le altre funzioni della classe